Journal of Systems Architecture 145 (2023) 103011

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

On the methodology of three-way structured merge in version control systems: Top-down, bottom-up, or both

Fengmin Zhu [a,d,1,2], Xingyu Xie [a,2], Dongyu Feng [a], Na Meng [e], Fei He [a,b,c,*]

[a] School of Software, Tsinghua University, China
[b] Key Laboratory for Information System Security, MoE, China
[c] Beijing National Research Center for Information Science and Technology, China
[d] Max Planck Institute for Software Systems, Germany
[e] Virginia Tech, USA

ARTICLE INFO

Keywords:
Version control systems
Three-way merging
Structured merging
Shifted code

ABSTRACT

Three-way merging is an essential component of version control systems. Despite the efficiency of the conventional line-based textual methods, syntax-based structured approaches have demonstrated benefits in improving merge accuracy. Prior structured merging approaches visit abstract syntax trees in a top-down manner, which struggles to identify and merge shifted code generally. This work introduces a novel methodology combining a top-down and a bottom-up visit of abstract syntax trees, which manipulates shifted code effectively and elegantly. Especially, we reduce the merge problem of ordered lists to computing a topological sort of strongly connected components of the constraint graph. Compared with four representative merge tools in 40,533 real-world merge scenarios, our approach achieves the highest merge accuracy while being 2.5 x as fast as a state-of-the-art structured merge tool.

1. Introduction

Thanks to the wide application of version control systems such as Git and SVN, *three-way merging* has become an indispensable task in contemporary software development. A *three-way merge scenario* $(base, left, right)$ consists of three versions of a program, where the two *variants* $left$ and $right$ are both evolved independently, possibly by different developers, from their ancestor $base$. A three-way merge algorithm integrates the changes made by the variants and produces a merged version called a *target*. When the two variants (i.e., branches) introduce changes that are contradicting, according to *three-way merge principles*, a *conflict* shall be reported, leaving the developers to manually resolve them. The three-way merge principles conservatively describe whether the changes could be correctly incorporated.

Unstructured merge is a mature merging approach that regards programs as a sequence of lines of plain text. Since the context-free syntax is neglected, merge accuracy is yet unsatisfying—studies [1,2] have shown the presence of *false conflicts* (i.e., the conflicts that should have been avoided), which increases the user burden of manual resolution. To enhance merge accuracy, *structured merge*, representing programs as *abstract syntax trees* (ASTs), has gained significant research interest in recent decades [2–8]. A structured merge algorithm takes a set of mappings between different program versions as input and computes a merged version as output. The mappings are obtained by AST differencing (also known as AST matching) algorithms [9–11].

To compute a target AST, a structured merge algorithm needs to traverse the input ASTs (i.e., $base$, $left$, and $right$). Prior approaches [2–8] all use a *top-down* order, which is quite natural and intuitive as it follows the structure of ASTs. Such a top-down AST comparison is usually restricted to be *level-wise*—only AST nodes at the same level (or depth) get compared, which makes it hard to detect if one piece of code is shifted into another, namely *shifted code* [12]. To identify shifted code in a top-down manner, one could search for the *largest common embedded subtree*. This problem, however, is known to be $\mathcal{NP}$-hard and difficult to approximate for general cases [13]. A more scalable approach is to employ *syntax-aware looking ahead* matching [12], but: looking ahead is only enabled for a few types of AST nodes; the maximum looking-ahead distance is short for efficiency considerations. The work in [12] focuses on the AST matching problem; how to correctly merge shifted code remains an issue.

Thinking oppositely, we find a *bottom-up* traversing order a better option. The key to detecting shifted code is to allow node mappings

* Corresponding author at: School of Software, Tsinghua University, China.
E-mail address: hefei@tsinghua.edu.cn (F. He).
1 Early revisions of this work were done when Fengmin Zhu was in Tsinghua University.
2 Fengmin Zhu and Xingyu Xie contributed equally.

https://doi.org/10.1016/j.sysarc.2023.103011
Received 5 February 2023; Received in revised form 31 July 2023; Accepted 6 October 2023
Available online 13 October 2023

across AST levels, which is natural and easier via a bottom-up manner. Meanwhile, top-down merging cannot handle across-level mappings, which means bottom-up merging is needed as follow-up of the matching phase. Sometimes, the bottom-up visit *alone* incurs redundant computations. As an extreme example, if $left$ is the same as $base$, meaning no changes are introduced on $left$, then by three-way merge principles, $right$ introduces unique changes and should be the target version—there is no need to further inspect any of their descendants as in a bottom-up manner. We fix this issue by bringing in top-down merging.

Combining a top-down pruning pass and a bottom-up pass, we present a novel three-way structured merge algorithm, where the trivial merge scenarios are processed in the former pass, and other nontrivial merge scenarios, which may involve shifted code, are carefully operated in the latter pass. Because our algorithm is non-backtracking, the time complexity is linear.

Like in JDime [2], we distinguish if the children list of an AST node can be "safely permuted" (i.e., the permutation preserves semantics). If not, the list is called *ordered* (e.g., a sequence of statements) and a good merge algorithm should preserve, as much as possible, the original occurrence order of the children in the merged versions, which we call *order-preservation*. We reduce this problem into computing strongly connected components and a topological sort of the directed acyclic graph formed by the components, which is solvable by linear-time graph algorithms, e.g., Tarjan's algorithm [14].

We implemented our approach as a structured merge tool called Mastery.[3] To measure its usability and practicality in real scenarios, we extract 40,533 merge scenarios from 78 open-source Java projects. We identified that shifted code occurs in 38.54% merge scenarios, and conducted experimental comparisons with four representative tools: JDime (structured), jFSTMerge (tree-based semistructured), IntelliMerge (graph-based semistructured), and GitMerge (unstructured). Our results show: (1) Mastery achieves the highest merge accuracy of 82.90%; (2) Mastery reports the fewest 9.33% conflicts and the fewest 7,057 conflict blocks, excluding radical IntelliMerge; (3) Mastery is about 2.5× as fast as JDime, and about 1.4× as fast as jFSTMerge. Our tool and evaluation data are publicly available: https://github.com/thufv/mastery/.

Contributions. To sum up, this paper makes the following contributions:

- We present a novel structured merge algorithm that visits ASTs in both a top-down and a bottom-up manner. The top-down pruning pass avoids a mass of redundant computations. The bottom-up pass makes it possible to handle shifted code elegantly and efficiently.
- The proposed merge algorithm is linear-time.
- We adapt a state-of-the-art tree matching algorithm to better fit our merge algorithm.
- We conduct comprehensive experiments on real-world merge scenarios. Results show that Mastery is competitive with state-of-the-art merge tools in the aspects of merge accuracy, the number of conflicts, and efficiency.

This paper is an extended version of a preliminary conference paper [15]. Compared to [15], this paper gives the following new materials. First, we elucidate how we utilize and adapt the state-of-the-art matching algorithm, GumTree [9], in order to make it better fit our merging algorithm. Second, we improve the ordered merging algorithm and report new experimental results. Whereas [15] adopts a topological sorting algorithm for merging ordered lists, in this paper, we compute strongly connected components in advance, trying to merge each component and concatenate the merged results of components in a topological sort of the components. This improvement makes Mastery able to handle trivial order-altering changes, e.g., swapping statements.

[3] Merging abstract syntax trees in a reasonable way.

Organization of this paper. We start in Section 2 with an introduction to three-way structured merging. In Section 3, we provide a bird's-eye view of our whole merging framework. Later on, we explain the technical details of our matching algorithm in Section 4 and merging algorithm in Section 5. We briefly present the tool implementation in Section 6 and carefully report our evaluation results in Section 7. Finally, we discuss related work in Section 8 and conclude the paper in Section 9.

2. Preliminaries

AST nodes. In structured merging, programs are represented as *abstract syntax trees* (ASTs), parsed from source files. An AST is a labeled rooted tree with four types of nodes, each annotated with the name of its production rule in the grammar, called its *label* (lbl).

$$\begin{array}{llll} \text{Node } v & ::= & \mathsf{Leaf}(lbl, x) & \text{(leaf)} \\ & \mid & \mathsf{Ctor}_k(lbl, v_1, \ldots, v_k) & (k\text{-ary constructor}) \\ & \mid & \mathsf{UList}(lbl, \{v_1, \ldots, v_n\}) & \text{(unordered list)} \\ & \mid & \mathsf{OList}(lbl, [v_1, \ldots, v_n]) & \text{(ordered list)} \end{array}$$

A *leaf node* represents a *lexical token* where the value x is the text of that token. A k-ary *constructor node* has exactly k children as its arguments. For instance, an if-statement – consisting of a Boolean condition, a true branch, and a false branch – is represented as a 3-ary constructor node. An arbitrary number of children is allowed in a *list node*, which is further divided into *unordered* – children can be safely permuted – and *ordered* (the opposite). For instance, a class member declaration list is unordered, while a statement list is ordered. The children of an unordered list node are denoted by a set $\{v_1, \ldots, v_k\}$, while the children of an ordered list node are denoted by a list $[v_1, \ldots, v_k]$ ($[]$ for an empty list). In case of merge conflicts, we introduce conflicting nodes in target ASTs. We assume that conflict nodes can never appear in any input AST (i.e., $base$, $left$, or $right$), as any previously reported conflicts should have already been resolved at the moment a new merge process is requested.

AST matching. A merging algorithm relies on a set of mappings between different versions of the programs to compute the merged version. The set of mappings between two ASTs T_1 and T_2 are represented by a *matching set* $\mathcal{M} = \{(u_i, v_i)\}_i$, where each pair (u_i, v_i) consists of two nodes $u_i \in T_1$ and $v_i \in T_2$. Intuitively, two mapped nodes are likely to be the same node in different versions: if we transform T_1 to T_2 by inserting and deleting nodes as few as possible, u_i will probably become v_i along the transformation. The mappings shall be *injective*—two nodes cannot be matched to the same node on the other AST simultaneously. Moreover, the matched nodes shall have the same label.

Definition 1 (*Shifted Code*)**.** Given two mappings $(u', v'), (u, v) \in \mathcal{M}$, if u is a child of u', whereas v is not a child (i.e., direct descendant) but a later descendant of v', then (u, v) is said a *shifted* mapping. Meanwhile, the code fragment corresponding to the subtree of v is called a *shifted code*.

For example, on the merge scenario shown in Fig. 1: in $left$, the code fragment of the `InfixExpr` is a shifted code and is shifted into a `CastExpr`; in $right$, the code fragment of the `ExprStmt` is a shifted code and is shifted into a `ForStmt`.

Three-way merge principles. A three-way merge algorithm must abide by a couple of principles, as presented in Table 1. To avoid repetition, the dual cases of rows 1, 3, and 4 are not displayed. The first two rules are applicable to all types of nodes. If a node is modified by exactly one of the variants, then the change is unique and itself gives the target (row 1). If a node is concurrently modified by both variants inconsistently, a conflict is reported, as the algorithm has no adequate information to decide which one to take (row 2). If the modifications are equal, then of course these changes are consistent and thus not

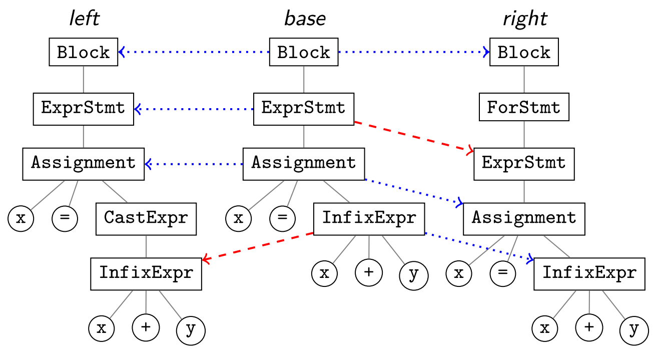


Fig. 1. A merge scenario with shifted code. (Shifted mappings are depicted as dashed arrows.)

Table 1
Three-way merge principles (dual cases omitted).

	Type	Version *base*	Version *left*	Version *right*	Target *T*	Explanation
1	Node	e	e	e'	e'	left-change
2	Node	e	e_L	e_R	conflict	inconsistent change
3	List	$e \in base$	e $\notin left$	e $\in right$	$e \notin T$	left-deletion
4	List	$e \notin base$	$e \in left$	$e \notin right$	$e \in T$ or conflict	left-insertion

conflicting. The last two rules are applicable for only (ordered and unordered) list nodes. If an element of *base* presents in exactly one of the variants, then it is regarded as being removed and will be excluded from the target list (row 3). In contrast, if a new node is introduced in exactly one of the variants, it will be inserted into the target list (row 4). For ordered lists, conflicts may occur if the insertion position is ambiguous. For unordered lists, the insertion position is arbitrary and thus not conflicting.

3. Framework overview

Fig. 2 presents the main process and the workflow of Mastery. The tree matcher and the tree merger only manipulate ASTs produced by a parser, and thus are language-agnostic. To enable structured merging for a certain programming language $\mathcal{L}$, a parser that builds ASTs from source code written in $\mathcal{L}$, and a pretty printer that outputs source code from ASTs are required to be integrated into the framework.

The framework takes three source files (they should be in the same language) as input, which forms a three-way merge scenario, and outputs another file as the merge result. The workflow consists of four sequential steps:

1. A parser translates the three source files into three ASTs, referred to as *base*, *left* and *right*.
2. A tree matcher (Section 4) compares and establishes mappings between the three ASTs. In our three-way tree matching algorithm, *base* is compared with *left* and *right* respectively, and each produces a matching set $\mathcal{M}_L$ and $\mathcal{M}_R$. The tree matching algorithm is an adaption of GumTree [9].
3. A tree merger (Section 5) then applies amalgamation on ASTs with the help of $\mathcal{M}_L$ and $\mathcal{M}_R$, and builds a target AST according to three-way merge principles. The algorithm is composed of two passes: the top-down pruning pass identifies *trivial* merge scenarios and processes them in advance; the bottom-up merging pass processes the remaining *non-trivial* merge scenarios using tree amalgamation algorithms. In case conflicts occur, special conflicting nodes are created to record the conflicting blocks.
4. A pretty printer traverses the target AST and outputs source code into a file. If the target AST contains conflict nodes, they will be displayed using standard *conflicting markers* (as in GitMerge).

4. Matching algorithm

This section presents our tree-matching algorithm. We adapt GumTree [9] to compute the set of mappings between two ASTs . We choose GumTree because it can search for cross-level mappings, which enables the detection of shifted code. We will first review its main steps in Section 4.1 and then introduce our adaptions in Section 4.2.

When it comes to a three-way merge scenario, we have in total three ASTs to compare: between which pairs shall we perform the underlying AST matching algorithms such as GumTree? Prior works [2,3,12] decided to do it between each pair of them, but we find this mechanism can sometimes lead to anti-intuitive results. To tackle this issue, we will propose a revised mechanism in Section 4.3.

4.1. GumTree overview

In a nutshell, the GumTree algorithm traverses the two input ASTs and searches for three kinds of mappings between them. An edit script can be deduced from the mappings using algorithms such as [16]. However, this step is not needed in our merge framework. To identify mappings, GumTree first performs a top-down visit of ASTs in decreasing height, finding *isomorphic* (equal) subtrees between them, each of which forms an *anchor mapping*. Then, it performs a bottom-up visit for identifying mappings between yet unmatched nodes. A *container mapping* is established if the two nodes have a large number of descendants (greater than a threshold) being matched, or, intuitively, being “quite similar”. Once a container mapping is found, *recovery mappings* are then established (e.g., using the RTED algorithm [17]) between some of their unmatched descendants.

GumTree is capable of identifying cross-level mappings, making the detection of shifted code available. Cross-level mappings can be established in both passes: (1) in the top-down pass, anchor mappings are established between two subtrees of the same height – not the same depth – thus cross-level isomorphic subtrees are probably regarded as matches; (2) in the bottom-up pass, container mappings are established if the two subtrees are similar, and again their depths may differ; the situation is the same for recovery mappings.

4.2. Our adaption: Monotone matchings

Classical unstructured merging approaches [18] exploit line-based matching. The matching task is to compute the longest common sequence of two sequences of lines, where a common element in the longest common sequence corresponds to a mapping. Thus, the matching between sequences must be non-crossing, i.e., the linear order of a sequence is preserved on the matching. As a natural extension of this non-crossing property, the mappings between trees ought to preserve the ancestor-descendant relationship. This is what we call *monotone*.

Definition 2 (*Monotonicity*)**.** A matching set $\mathcal{M}$ is *monotone* if the following holds: for every $(s,t) \in \mathcal{M}$, if there exists $u \in s$ (u is a descendant of s) such that $(u,v) \in \mathcal{M}$ for some node v, then $v \in t$ (v is a descendant of t).

To compute a monotone matching set, we use an ad-hoc filter strategy. Every time a new mapping is identified, we check if the monotonicity property is preserved by definition. If it is preserved, the new mapping is accepted, otherwise, rejected. Noting the fact that “for two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder and after y in the postorder”, the monotonicity (Definition 2) could be described by the order of pairs of numbers. Thus, by precomputing the preorder and postorder of the nodes and caching previously-computed results, the checking procedure takes only constant time. In other words, it does not increase the worst-case time complexity of GumTree.

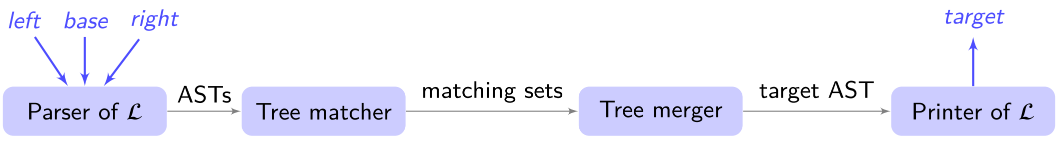


Fig. 2. Workflow of our merge framework Mastery.

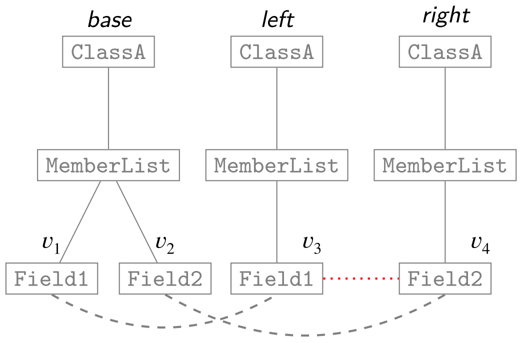


Fig. 3. A merge scenario $(base, left, right)$ with established mappings (dashed or dotted lines). The red dotted line shows a mapping established by comparing $left$ and $right$. Assume the similarity (measured by a heuristic cost function) between `Field1` and `Field2` exceeds a given threshold.

4.3. Which ASTs to compare?

Given a three-way merge scenario $(base, left, right)$, which pairs of ASTs should be compared? A trivial solution (used by [2,3,12]) might be: to compare each pair of them, i.e., $(base, left)$, $(base, right)$, $(left, right)$. Each tree matching problem yields a matching set, denoted by $\mathcal{M}_{BL}$, $\mathcal{M}_{BR}$ and $\mathcal{M}_{LR}$. This looks like a reasonable method at first sight, however, we recognize an exhibition of anti-intuitive circumstances. For example, it could be the case that there exist nodes $b \in base$, $l \in left$ and $r \in right$ such that $(b, l) \in \mathcal{M}_{BL}$ and $(l, r) \in \mathcal{M}_{LR}$, meanwhile $(b, r) \notin \mathcal{M}_{BR}$. Typically, it is expected that the matching relation is *transitive*, say given $(b, l) \in \mathcal{M}_{BL}$ and $(l, r) \in \mathcal{M}_{LR}$, we should also have $(b, r) \in \mathcal{M}_{BR}$. However, the above case violates the transitivity.

Fig. 3 presents a merge scenario $(base, left, right)$. Neglecting the mappings, one can recognize that both `Field1` and `Field2` are deletions – by row 3 of Table 1. Therefore, the expected merge result is simply a class definition without any field. We now take a closer look at the established mappings: v_1 is matched with v_3 (isomorphic), and v_3 is matched with v_4 (though not isomorphic, they are mapped because of their high similarity). However, v_1 is not matched with v_4, which violates the transitivity of the matching relation. Instead, we see v_2 is matched with v_4, as they are isomorphic. In the subsequent AST amalgamation pass, we have to answer a tricky question: shall we perform merging on the merge scenario (v_1, v_3, v_4)? If yes, then by row 1 of Table 1, `Field2` gives the merge result of the above merge scenario. Later, `Field2` will be included—however, it should be regarded as a deletion. If no, then there would be no way to figure out that `Field1` is a deletion. In either case, the amalgamation algorithm will not give the expected merge.

To resolve the above dilemma, we propose to *exclude* the comparison between $left$ and $right$. In the above example, that means the mapping (v_3, v_4) (red dotted line) no longer exists and by three-way merge principles, the expected merge will be successfully produced. In this way, (b, l, r) is regarded as a "proper" merge scenario (so that amalgamation is performed on it) if and only if $(b, l) \in \mathcal{M}_{BL}$ and $(b, r) \in \mathcal{M}_{BR}$. Our rationale is that, since the two variants both evolve from $base$, they each should be compared with $base$ for identifying the changes they made. Conversely, the two variants are *neither* evolved from each other. In reality, they are usually developed *independently*, and sometimes the developers may even not know each other (common in open-source projects). Thus, the two variants are not that related. As the essence of the three-way merge is to *integrate changes* – rather than to *recognize differences* – we find it unnecessary to compare $left$ and $right$. One may concern that the above strategy can sometimes miss a mapping that should be established between the two variants (since they are not compared to each other at all). To identify such mappings and to preserve the transitivity in the meantime, we could additionally compare and establish mappings between the nodes of the two variants that are not matched with any node of $base$.

5. Merging Algorithm

Given a three-way merge scenario $(base, left, right)$, our merging algorithm accepts two matching sets from the matching algorithm as input—$\mathcal{M}_L$ the matches between $base$ and $left$, and $\mathcal{M}_R$ the matches between $base$ and $right$. The algorithm generates a new tree, namely a *target AST*, as the merge result. Merging is performed on the matched nodes only, as unmatched nodes are assumed to have no relation. In the case of three-way merging, the two variants should match the base version. Formally, a merge scenario (b, l, r) is said *proper* if $(b, l) \in \mathcal{M}_L$ and $(b, r) \in \mathcal{M}_R$. The merge algorithm only needs to manipulate proper merge scenarios. Non-proper merge scenarios can be safely omitted because they are regarded as deletions and thus do not appear in the target.

Fig. 4 presents the high-level workflow of our merge algorithm. It consists of a *top-down* pass followed by a *bottom-up* one. In the top-down pass (see Section 5.1 for details), input ASTs get traversed in pre-order, and any *trivial* merge scenario – any two of the three versions are equal – is processed immediately. Meanwhile, we collect other *non-trivial* proper merge scenarios in the list $\mathcal{S}$. In the bottom-up pass, the merge scenarios in $\mathcal{S}$ get processed in the reverse order, i.e., in post-order. Since matched nodes have the same label, the three versions in a proper merge scenario must be homogeneous—they must all be leaf nodes, constructor nodes, unordered list nodes, or ordered list nodes.

A challenging problem in the bottom-up pass is that: a sub-scenario (b, l, r) may *not be proper*, say b and l do not match. Even though it is rational to assume b matches *some descendant* of l (or else we simply report a conflict), which happens when l has shifted code. We encode this condition as a *relevant-to* relation: u is relevant to v, written $u \simeq v$, iff there exists a descendant $w \in v$ such that u matches w. With this notion, the assumption we make on a sub-scenario (b, l, r) is given by $b \simeq l \wedge b \simeq r$. Merging such a sub-scenario requires us to take shifted code into account. We will present this algorithm in Section 5.3.

The merge result of the merge scenario (b, l, r) is recorded in a map $\mathcal{R}$ so that the algorithm can query it later on demand. Instead of using the entire merge scenario (b, l, r) as the index (or key) for the map $\mathcal{R}$, realizing that any node b of $base$ appears in at most one merge scenario (by the injectivity of the matching sets), we simply use b as the index. In the end, the target AST of the top-most merge scenario $(base, left, right)$ is obtained by querying $\mathcal{R}(base)$.

5.1. Top-down merging

In the top-down pass, we visit $base$ in a descendant recursive manner by invoking `TopDownVisit` (Algorithm 1). This function returns all non-trivial merge scenarios that need processing later in the bottom-up

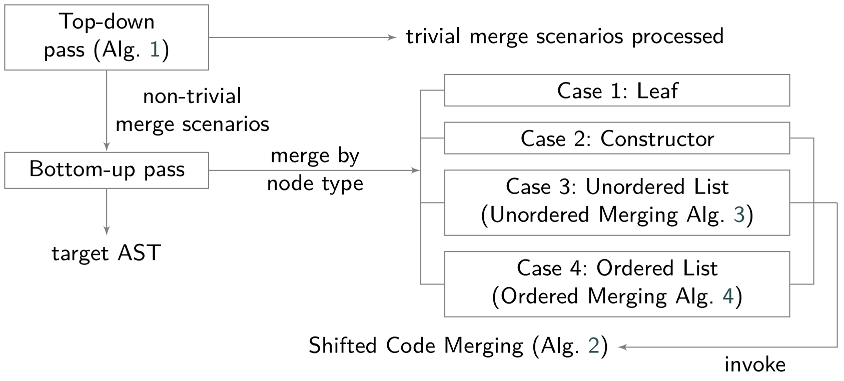

Top-down pass (Alg. 1)
trivial merge scenarios processed
non-trivial merge scenarios
Bottom-up pass
merge by node type
target AST
Case 1: Leaf
Case 2: Constructor
Case 3: Unordered List (Unordered Merging Alg. 3)
Case 4: Ordered List (Ordered Merging Alg. 4)
Shifted Code Merging (Alg. 2)
invoke

Fig. 4. High-level workflow of our merge algorithm.

Algorithm 1: Top-down pruning pass

```
1 Function TopDownVisit(b: Node):
2     $S \leftarrow []$;
3     if $\exists l \in left, r \in right : (b,l) \in \mathcal{M}_L \wedge (b,r) \in \mathcal{M}_R$ then
4         if $b = r$ then $\mathcal{R}(b) \leftarrow l$; return [];
5         if $b = l$ or $l = r$ then $\mathcal{R}(b) \leftarrow r$; return [];
6         $S$ += $(b,l,r)$;
7     foreach child $c$ of node $b$ do
8         $S$ ++= TopDownVisit($c$);
9     return $S$;
```

Algorithm 2: Shifted Code Merging

```
1 Function IssueShifted(b: Node, l: Node, r: Node):
2     let $l', r'$ be nodes s.t. $(b,l') \in \mathcal{M}_L, (b,r') \in \mathcal{M}_R$;
3     if $l' = l \wedge r' = r$ then return $\mathcal{R}(b)$;
4     if $l' \neq l \wedge r' = r$ then return $l[\mathcal{R}(b)/l']$;
5     if $r' \neq r \wedge l' = l$ then return $r[\mathcal{R}(b)/r']$;
6     if $l[\mathcal{R}(b)/l'] = r[\mathcal{R}(b)/r']$ then return $l[\mathcal{R}(b)/l']$;
7     return Conflict($l, r$);
```

pass. We use a list S to collect them. For short, we use two notations: S += e for appending an element e to S, and S ++= S' for appending all elements in S' to S. Upon traversing, if any trivial merge scenario is encountered, we immediately store the target AST in $\mathcal{R}$ and prune any further visit of its sub-scenarios by returning an empty list (lines 4–6). Otherwise, we proceed to collect merge scenarios recursively (lines 8–9).

5.2. Two base cases in bottom-up merging

The first case, all nodes are leaf nodes, is the base case of our merge algorithm. This case is straightforward by three-way merge principles: we either take the only-changed variant as the target (by row 1 of Table 1) or report a conflict due to the inconsistent changes (by row 2).

The second case, merging constructor nodes of the same label and arity, gives a constructor node of that label and arity too, and each child node is recursively merged from the sub-scenarios formed by the children at the corresponding index. Merging list nodes gives list nodes too, and it contains elements recursively merged from certain sub-scenarios drawn from the elements in the input lists (see Section 5.4 and Section 5.5 for details).

5.3. Shifted code merging

Shifted code may exhibit in any type of node (except leaf node) in merge scenarios, which indicates the method of merging shifted codes is a really important utility. Algorithm 2 presents a unified algorithm for dealing with shifted code. It requires $b \simeq l \wedge b \simeq r$. Merging is performed according to where the shifted code involves:

(no shifting) If (b,l,r) is proper, then we simply query $\mathcal{R}$ (line 3).

(left-shifting) If b matches r but not l, then there exists a l' such that it is shifted into l. To integrate this shifting, we first make a copy of l and replace l' with the merge result of (b,l',r') i.e., $\mathcal{R}(b)$ (line 4). The notation $u[w/v]$ gives an updated tree by replacing a subtree v with w on u.

(right-shifting) Line 5 is symmetric to the above case.

(consistent-shifting) If both variants involve shifted code, the only circumstance we can safely merge is when they yield the same result (line 6).

(inconsistent-shifting) Otherwise, report a conflict (line 7).

Consider the example in Fig. 1. When merging the merge scenario consisting of the three `Assignments`, `CastExpr` from $left$, `InfixExpr` from $base$ and `InfixExpr` from $right$ form the arguments b, l, r in Algorithm 2. The result is computed by taking the subtree of `CastExpr` and replacing its subtree of `InfixExpr` with the target one (by line 4). In merging the top-most merge scenario, the result is computed from the subtree of `ForStmt` by replacing its child `ExprStmt` with the target of `ExprStmts` (by line 5). In this way, the shifted changes made by the two variants are integrated.

5.4. Unordered merging

Let B, L, and R respectively be the set of elements of three unordered list nodes that form a merge scenario. The goal of unordered merging is to compute a set of elements T – without worrying about the order – that should appear in the target list. These elements are classified as follows:

(shifting) If an element $b \in B$ satisfies $b \simeq l \wedge b \simeq r$ for some $l \in L$ and $r \in R$, then the merge result is obtained by invoking `IssueShifted(b, l, r)`.

(left/right-insertion) If an element of L or R is not related to any element of B, then by row 4 of Table 1 it is an insertion.

(left/right-deletion-change conflict) If an element b satisfies, for example (dual case is similar), $b \simeq r$ for some $r \in R$, then it is a left-deletion (thus not included in T) when $b = r$; and a left-deletion-change conflict when $b \neq r$.

Algorithm 3: Unordered merging

```
Function Unordered(B: Set, L: Set, R: Set):
    $T \leftarrow \emptyset$;
    foreach $b \in B$ do
        if $\exists l \in L, r \in R : b \simeq l \wedge b \simeq r$ then
            $T \leftarrow T \cup \{$IssueShifted(b, l, r)$\}$;
            mark $l$, $r$ as "visited";
        else if $\exists l \in L : b \simeq l$ then // right case is symmetric
            if $b \neq l$ then $T \leftarrow T \cup \{\mathsf{Conflict}(l, \varepsilon)\}$;
            mark $l$ as "visited";
    $T \leftarrow T \cup \{e \mid e \in L \cup R, e$ is not "visited"$\}$;
    return $T$;
```

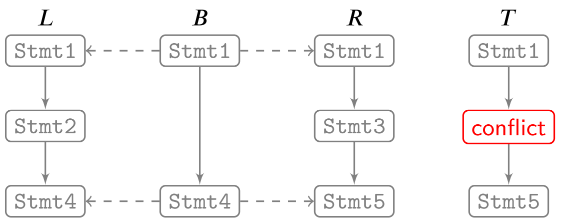


Fig. 5. A conflicting merge scenario of three statement-lists (B, L, R) with the target T. A dashed edge between two statements indicates they are matched.

The above is realized as Algorithm 3: First, traverse the elements in B and collect any shifting (line 4) or left-deletion-change conflict (line 7, right-deletion-change conflict is symmetric) in T. Meanwhile, mark every relevant left/right element as "visited" (lines 6 and 9). Then, all elements yet not marked must be left/right-insertions: thus insert them into T (line 10).

5.5. Ordered merging

Merging ordered lists is more complex than merging unordered lists in that the elements of the target list should be in an order preserving the original occurrence order of associated elements in the merge scenario; and it is necessary to decide whether such an order uniquely exists—if not, to fit the merge algorithm into a conservative setting, conflicts shall be reported as well. For example, Fig. 5 depicts a merge scenario where the three ordered lists B, L, and R each represent the statements inside a block. For clearance, we display the elements of each list in a linked chain rather than an AST. A dashed edge between two nodes indicates they are matched. By three-way merge principles, `Stmt1` and `Stmt5` should be included in the target list T: since `Stmt1` precedes `Stmt4` in B, `stmt1` must also precede `Stmt5` in T. Next, both `Stmt2` and `Stmt3` should be included in T as they are insertions. However, their insertion order is ambiguous: no information can tell if `Stmt2` should precede `Stmt3` or the other way around; indeed, we just know either must come in between `Stmt1` and `Stmt5`. Due to the ambiguous insertion order, a conflict has to be reported.

In some extreme cases, strictly preserving the order is not desired, particularly, when the change introduced is just permutating the order. For example, in Fig. 6, `Stmt1` and `Stmt2` are swapped in L. By three-way merge principles, since only L introduces a swap-change, the target T should accept this swap-change.

Order-preserving. Before presenting the merge algorithm, we first need a formal interpretation of "preserving the original occurrence order". Since the occurrence order is a partial order relation, it is natural to regard the three ordered lists in the merge scenario (B, L, R) as three ordered sets $\langle B, \lhd_B\rangle$, $\langle L, \lhd_L\rangle$ and $\langle R, \lhd_R\rangle$. The occurrence order

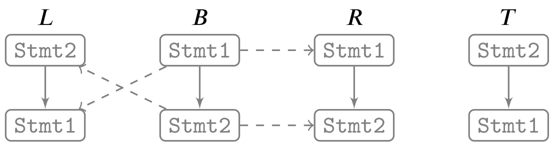


Fig. 6. A swap-change merge scenario of three statement-lists (B, L, R) with the target T. A dashed edge between two statements indicates they are matched.

Algorithm 4: Ordered merging

```
Function Ordered(B: List, L: List, R: List):
    $\Phi \leftarrow$ GenConstraints(B, L, R);
    Represent $\Phi$ as a directed graph $G_\Phi = \langle V, E\rangle$;
    $C \leftarrow$ Tarjan($G_\Phi$) ; /* a topological sort of SCCs */
    for $i \leftarrow 2..|C|$ do
        if there is no edge $(u, v) \in E$ s.t. $u \in C[i-1] \wedge v \in C[i]$ then
                /* multiple topological sorts */
            return "conflict" ;
    $T \leftarrow []$;
    for $i \leftarrow 1..|C|$ do
        if $\pi_B|_{C[i]} \neq \pi_L|_{C[i]} \wedge \pi_B|_{C[i]} \neq \pi_R|_{C[i]} \wedge \pi_L|_{C[i]} \neq \pi_R|_{C[i]}$
         then
                /* three unique constraints in a SCC */
            return "conflict" ;
        if $\pi_B|_{C[i]} = \pi_L|_{C[i]}$ then $X \leftarrow R$ ;
        else if $\pi_B|_{C[i]} = \pi_R|_{C[i]}$ then $X \leftarrow L$ ;
        else $X \leftarrow L$ ;            /* $\pi_L|_{C[i]} = \pi_R|_{C[i]}$ */
        foreach $u$ in the order $\pi_X|_{C[i]}$ do $T$ += $u$ ;
    return $T$;
```

relation is denoted by $\lhd_X$ (for $X \in \{B, L, R\}$), formally defined as $X[i] \lhd_X X[j] \iff i < j$.

Let S be the set of elements that should appear in the target list T, computed by `Unordered(`B, L, R`)`. In Section 5.4, we classify these elements into: shifting, left/right-insertion, and left/right-deletion-change conflict. Each element is computed from certain elements in the input merge scenario. For example, if $t^* \in S$ is a left-insertion, then t^* is a copy of some $l^* \in L$. We encode their relationship as partial functions $\pi_B : B \rightharpoonup T$, $\pi_L : L \rightharpoonup T$ and $\pi_R : R \rightharpoonup T$. For the above example, we let $\pi_L(l^*) = t^*$. We encode the relationship as partial functions $\pi_B : B \rightharpoonup T$, $\pi_L : L \rightharpoonup T$ and $\pi_R : R \rightharpoonup T$, associating an element in the input merge scenario with the corresponding element in the target list. For example, if $t^* \in S$ is a left-insertion, then t^* is a copy of some $l^* \in L$, thus we let $\pi_L(l^*) = t^*$.

Definition 3 (*Order-preserving*)**.** We say an ordered list T is *order-preserving* w.r.t. (B, L, R) if T is a permutation of S = `Unordered(`B, L, R`)` such that π_B, π_L, and π_R are monotone. A partial function $f : X \rightharpoonup Y$ is said *monotone* if for every $x_1, x_2 \in X$ such that $f(x_1)$ and $f(x_2)$ are both defined, $x_1 \lhd_X x_2$ entails $f(x_1) \lhd_Y f(x_2)$.

In the above, we use the monotonicity condition to formalize our requirement of "preserving the original occurrence order".

Algorithm. The main goal of the algorithm (4) is to solve an order-preserving list and to decide the uniqueness of such lists. We compute an order-preserving list via constraint-solving—the constraints encode the monotonicity condition for the target list T by Definition 3. Technically each constraint has the form $e_1 \lhd e_2$, meaning "e_1 precedes e_2 in T". We propose an algorithm `GenConstraints` (Algorithm 5) to produce them by traversing the elements of B, L, and R in their occurrence order. We explain its correctness by the following lemma:

Algorithm 5: Constraints generation

```
Function GenConstraints(B: List, L: List, R: List):
    foreach $b \in B$ do
        if $\exists l \in L, r \in R : b \simeq l \wedge b \simeq r$ then
            $t \leftarrow$ IssueShifted(b, l, r);
            $\pi_B \leftarrow \pi_B \cup \{b \mapsto t\}$, $\pi_L \leftarrow \pi_L \cup \{l \mapsto t\}$,
              $\pi_R \leftarrow \pi_R \cup \{r \mapsto t\}$;
            mark $l$, $r$ as "visited";
        else if $\exists l \in L : b \simeq l$ then
            $t \leftarrow$ IssueShifted(b, l, r);
            $\pi_B \leftarrow \pi_B \cup \{b \mapsto t\}$, $\pi_L \leftarrow \pi_L \cup \{l \mapsto t\}$;
            mark $l$ as "visited";
        else if $\exists r \in R : b \simeq r$ then
            $t \leftarrow$ IssueShifted(b, l, r);
            $\pi_B \leftarrow \pi_B \cup \{b \mapsto t\}$, $\pi_R \leftarrow \pi_R \cup \{r \mapsto t\}$;
            mark $r$ as "visited";
    foreach $l \in L$, $l$ is not "visited" do
        $\pi_L \leftarrow \pi_L \cup \{l \mapsto l\}$;
    foreach $r \in R$, $r$ is not "visited" do
        $\pi_R \leftarrow \pi_R \cup \{r \mapsto r\}$;
    $\Phi \leftarrow \emptyset$;
    for $X \in \{B, L, R\}$ do
        for $i \in 2$ to $|X|$ do
            $\Phi \leftarrow \Phi \cup \{\pi_X(X[i-1]) \lhd \pi_X(X[i])\}$;
    return $\Phi$;
```

Lemma 1. *Given a merge scenario (B, L, R) of three ordered lists. If Φ is the constraint set returned by Algorithm 5, then the set of satisfying solutions to Φ is equivalent to the set of order-preserving lists, i.e., every satisfying solution of Φ is an order-preserving list and vice versa.*

Proof. Let T be an order-preserving list. The monotonicity requirement is translated into the following logical formula:

$$\begin{aligned}\Theta = &\{\pi_B(B[i]) \lhd_T \pi_B(B[j]) \mid i < j, \pi_B(B[i]) \text{ and } \pi_B(B[j]) \text{ are defined}\}\\ &\cup \{\pi_L(L[i]) \lhd_T \pi_L(L[j]) \mid i < j, \pi_L(L[i]) \text{ and } \pi_L(L[j]) \text{ are defined}\}\\ &\cup \{\pi_R(R[i]) \lhd_T \pi_R(R[j]) \mid i < j, \pi_R(R[i]) \text{ and } \pi_R(R[j]) \text{ are defined}\}\end{aligned}$$

Therefore, it suffices to show:

$$\bigwedge_{\theta\in\Theta} \theta \iff \bigwedge_{\varphi\in\Phi} \varphi. \qquad (1)$$

On the one hand, by comparing lines 20–22 of the algorithm (i.e., the definition of Φ) to the definition of Θ, we observe that $\Phi \subseteq \Theta$, which implies

$$\bigwedge_{\theta\in\Theta} \theta \implies \bigwedge_{\varphi\in\Phi} \varphi. \qquad (2)$$

On the other hand, Θ is the transitive closure of Φ over the relation $\lhd_T$, we thus have

$$\bigwedge_{\varphi\in\Phi} \varphi \implies \bigwedge_{\mu\in\Theta\setminus\Phi} \mu,$$

which implies

$$\bigwedge_{\varphi\in\Phi} \varphi \implies \bigwedge_{\theta\in\Theta} \theta. \qquad (3)$$

By Eq. (2) and Eq. (3), we see Eq. (1) holds. □

We propose an algorithm GenConstraints to produce them by traversing the elements of B, L, and R in their occurrence order, e.g., we generate the constraint "$\pi_B(b_1) \lhd \pi_B(b_2)$" for $b_1 \lhd_B b_2$.

Now let us move back to Algorithm 4. Let Φ be the set of constraints returned by Algorithm 5 (line 2). We represent the constraints as a directed graph (line 3) $G_\Phi = \langle V, E\rangle$, where: (1) the set of vertices are the elements of Unordered(B, L, R), i.e., $V = S$, and (2) for each constraint $(e_1 \lhd e_2) \in \Phi$, let $(e_1, e_2) \in E$ be an edge of G_Φ. It is well-known from graph theory that: there is a one-one correspondence between a topological sort of G_Φ and a satisfying solution of Φ, which further implies that: there is a one-one correspondence between an order-preserving list and a topological sort of G_Φ.

The first algorithm that comes to mind is perhaps a topology sort algorithm such as Kahn's algorithm [19]. However, it is too strict to handle cases like Fig. 6. A more flexible way is to compute strongly connected components (SCCs) first, try to merge each SCC on its own, then try to find a unique topological sort of the directed acyclic graph formed by the SCCs.

Algorithm 4 shows the details. In line 4, we facilitate Tarjan's strongly connected components algorithm to compute SCCs. Contracting each SCC as a supernode, we get a *component graph*, which is directed acyclic. As a byproduct, Tarjan's algorithm also produces a topological sort of component graph. We store the computed topological sort of SCCs as a one-based list in C.

Lines 5 – 7 check if the topological sort is unique, which is equivalent to that there is an edge between each pair of adjacent SCCs in the component graph. And, this is equivalent to check that whether there are two connected nodes respectively from the adjacent SCCs (line 6, highlighted).

Then we try to merge each SCC and concatenate the merged results sequentially in T (lines 8–15). We use $\pi_B|_{C[i]}$ to denote the constraints generated for SCC $C[i]$ from B. The notations for L and R are similar. Lines 10 – 11 consider a case that is too complicated to deal with: the order constraints generated from the three versions are pairwise distinct. We conversely report a conflict for this case. If it is not the case, we can merge the SCC along the order constraints of one version. Lines 12 – 14 choose which version to obey: (1) if the constraints of B and L are equal, which means only R possibly introduces changes, we choose R (line 12); (2) symmetrically, if the constraints of B and R are equal, which means only L possibly introduces changes, we choose L (line 13); (3) if L and R introduce the same changes, we adopt the changes (line 14). The concatenating operation takes place in line 15. When the concatenation finishes and no conflict has been found, line 16 returns the target list.

The ordered merge algorithm is linear because both Tarjan's algorithm and the generation of constraints are linear. Moreover, the other algorithms mentioned before are also linear, thus:

Theorem 1. *The time complexity of the entire structured merge algorithm is linear (to the size of the input merge scenario).*

6. Implementation

We implemented the proposed approach as a structured merge framework, Mastery, written in Java. This framework consists of four modules:

1. a parser that translates input source files into ASTs;
2. a tree matcher that generates mappings between different program versions, using an adapted GumTree [9] algorithm;
3. a tree merger that computes the target AST, following the algorithms presented in Section 5;
4. a pretty printer that outputs the formatted code from the merged AST.

Mastery currently supports merging Java programs. We rely on JavaParser,[4] a mature open-source parsing library for the Java language, to build ASTs from source code and pretty print Java source

[4] https://javaparser.org/

code from ASTs. To integrate with our merge algorithm, we convert the ASTs generated by JavaParser into our AST data structure (defined in Section 2).

7. Evaluation

To measure the usability and practicality of our approach in real-world scenes, we extract 40,533 merge scenarios from 78 Java open-source projects hosted on GitHub, and then conduct a series of experimental evaluations to answer the following research questions:

RQ1: How often does shifted code occur in real-world merge scenarios?
RQ2: What is the merge accuracy of Mastery when compared to state-of-the-art merge tools?
RQ3: How many merge conflicts are reported by these tools?
RQ4: What is their performance from the perspective of runtime?

7.1. Experimental setup

To select realistic and representative merge scenarios as our evaluation dataset, we seek the top 100 most popular open-source Java projects hosted on GitHub[5]; and then exclude any non-software-project (e.g., tutorials). On the remaining 78 projects, we extract merge scenarios via an analysis of their commit histories:

1. Using standard `git` commands, we extract all *merging commits* with its two parents, and their *base commit*s. The corresponding Java source files in the three version commits are collected as a merge scenario. In terms of three-way merge, the one in the base commit is the base version, and the source files in the two parent commits are the variants (i.e. left and right) version. The corresponding source file in the merge commit itself is marked as the *expected* version and will be used as the *ground truth*.
2. Usually, not all source files are changed from one version to another—only a few are affected. Furthermore, given the base, left and right versions of a source file, if any of the two are equivalent, by three-way merge principles, we immediately know the target version without performing the merge. This kind of merge scenarios is not worth evaluating, as any merge approach should produce the correct output. To better examine the differences between the merge tools, we instead remove this kind of merge scenarios, that is, we only collect the three versions of a source file that are pairwise distinct. To achieve this, we use `git diff` to distinguish whether two files are distinct.
3. Some source files cannot be parsed correctly, e.g. they include unresolved conflicts. We remove them as structured approaches assume the input files must have valid syntax (checked by JavaParser).

Further, we evaluate Mastery by comparing with four state-of-the-art merge tools:

- JDime [2], a state-of-the-art structured merge tool,
- JFSTMerge [6], an improved version of FSTMerge, a well-known tree-based semistructured merge tool,
- IntelliMerge [20], a refactoring-aware graph-based semistructured merge tool, and
- GitMerge, the default merging algorithm in Git.

All experiments were conducted on a workstation with AMD EPYC 7H12 64-Core CPU and 1TB memory, running Ubuntu 20.04.3 LTS.

[5] According to the following list, until July 12, 2021: https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Java.md.

7.2. Frequency of shifted code (RQ1)

In this evaluation, we aim to understand how often shifted code presents in practice. To calculate the frequency of shifted code, we use the state-of-the-art AST differencing tool, GumTree [9], to compute matchings among $base$, $left$, and $right$. Note that we do not need any merge tool for this evaluation. Then, on the computed matchings, we count the number of shifted mappings (i.e. shifted code) according to Definition 1.

Fig. 7 presents how many merge scenarios in each studied project *involve* shifted code, meaning at least one shifted mapping is detected. Among the 40,533 merge scenarios, we find 15,620 merge scenarios involve shifted code—the frequency is 38.54%. In those merge scenarios, we detect 90,982 shifted mappings—on average 2.24 shifted mappings per merge scenario.

7.3. Taxonomy of results

To understand the behavioral performance of the merge tools, we classify a merged result (for each merge scenario of each tool) into one of the following four categories:

- *expected*: the merged file and the expected version (the ground truth) are *syntactically equivalent*, i.e., their ASTs are isomorphic to each other (allowing permutations of elements in an unordered list node);
- *unexpected*: the merged file is conflict-free and is nonequivalent to the expected version;
- *conflicting*: there is at least one conflicting block in the merged file;
- *failed*: either the tool crashes or the execution exceeds the time limit 300 s.

Table 2 shows the distribution of the results of the five tools. The percentage of each kind of result is visualized in Fig. 8. JDime and IntelliMerge failed on a considerable number of scenarios, mainly caused by their implementation bugs. IntelliMerge represents each program version as a *program element graph*, and adopts a radical merging algorithm to construct the merged graph, which may violate the three-way merge principles when a deletion happens. Thus, it tends to produce more unexpected results. For GitMerge, only 1.95% results are unexpected. To understand this phenomenon, we have to notice that all projects use GitMerge as default. If GitMerge does not report any conflict, the merged codes will usually become the *ground truth* in our evaluation, without being reviewed by the developers. Thus, the expected version is a kind of *biased* ground truth in favor of GitMerge. This finding is consistent with a previous work [20].

7.4. Merge accuracy (RQ2)

In this evaluation, we study the merge *accuracy*, calculated as the percentage of expected results, of the five tools on our dataset. As shown in Table 2, Mastery achieves the highest accuracy of 82.90% among all tools. Compared to JDime, Mastery gains 3.46% higher accuracy. Among the 2,028 scenarios where Mastery's results are expected whereas JDime's are not, we find 49.65% involves shifted code—11.11% higher than the overall frequency.

To illustrate the capacity of Mastery on handling shifted codes, consider the merge scenario depicted in Fig. 9 (extracted from our dataset, slightly simplified for readability): the method call expression `invocation.getAttachments(...)` is shifted into a type cast expression (cast to `String` type) in the right version, and the left version modifies the argument of the method call (the prefix `Constants` is deleted). Mastery produces the desired merge, that is, to collaborate the above two changes, whereas JDime reports a conflict.

Recall that IntelliMerge adopts a radical merging strategy, which makes its accuracy only 24.12%. As discussed in Section 7.3, the ground

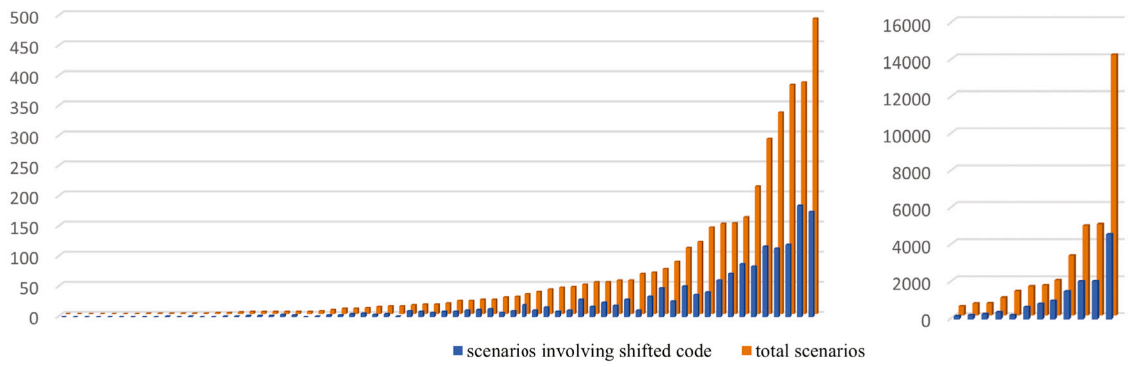


Fig. 7. Distribution of shifted code in each project. The projects are sorted by their number of merge scenarios in ascending order. We split them into two subfigures according to if the number of merge scenarios is less than 500 (left) or not (right).

Table 2
Distribution of the merged results.

Tool	Expected		Unexpected		Conflicting		Failed	
	Number	Accuracy	Number	Percentage	Number	Percentage	Number	Percentage
Mastery	**33602**	**82.90%**	3148	7.77%	3782	9.33%	1	0.00%
JDime	32200	79.44%	2406	5.94%	4538	11.20%	1389	3.43%
jFSTMerge	30062	74.17%	3837	9.47%	6626	16.35%	8	0.02%
IntelliMerge	9777	24.12%	24553	60.58%	**3442**	**8.49%**	2761	6.81%
GitMerge	30643	75.60%	**791**	**1.95%**	9099	22.45%	**0**	**0.00%**

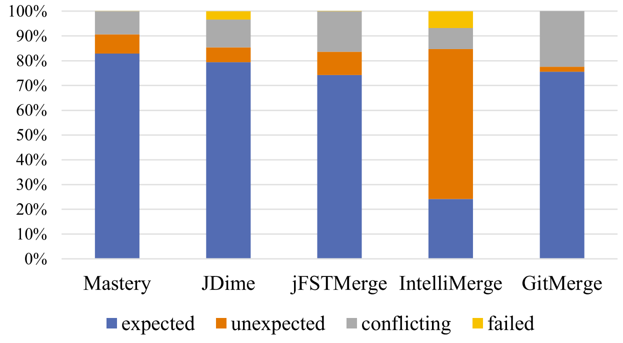
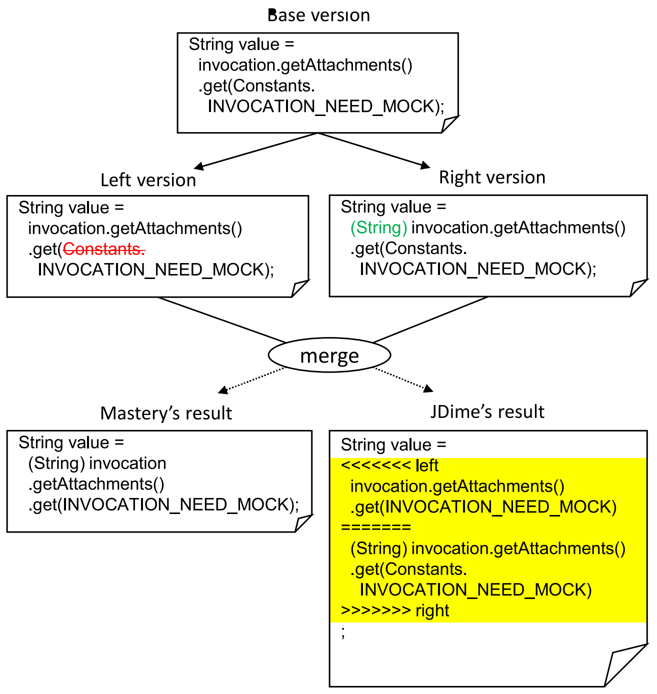


Fig. 8. Distribution of the merging results.



Base version:
```
String value =
  invocation.getAttachments()
  .get(Constants.
    INVOCATION_NEED_MOCK);
```

Left version:
```
String value =
  invocation.getAttachments()
  .get(~~Constants.~~
    INVOCATION_NEED_MOCK);
```

Right version:
```
String value =
  (String) invocation.getAttachments()
  .get(Constants.
    INVOCATION_NEED_MOCK);
```

Mastery's result:
```
String value =
  (String) invocation
  .getAttachments()
  .get(INVOCATION_NEED_MOCK);
```

JDime's result:
```
String value =
<<<<<<< left
  invocation.getAttachments()
  .get(INVOCATION_NEED_MOCK)
=======
  (String) invocation.getAttachments()
  .get(Constants.
    INVOCATION_NEED_MOCK)
>>>>>>> right
;
```

Fig. 9. A merge scenario from commit 9f5cc83 of project dubbo, where an expression gets shifted. Mastery's result is expected, while JDime's is conflicting.

truth is biased in favor of GitMerge, causing the accuracy of GitMerge to be even larger than jFSTMerge.

The scenarios where GitMerge produces unexpected or conflicting results are of special interest to us—in these merge scenarios, the expected versions (i.e., the merged versions in Git histories) must have been reviewed by the developers. If we consider only these 9,890 scenarios, the accuracy of the five tools except GitMerge are:

Mastery	JDime	jFSTMerge	IntelliMerge
33.35%	31.17%	17.26%	6.98%

Mastery still achieves the highest accuracy.

7.5. Reported conflicts (RQ3)

In this evaluation, we measure the number of conflicts reported by the five tools on all merge scenarios. The numbers and percentages of conflicting results are listed in Table 2. We also count the number of conflicting blocks (i.e. conflicting hunks) reported by the five tools, which are depicted in Fig. 10. Especially, IntelliMerge's radical strategy ignores three-way merge principles, making it achieve the lowest in both metrics. The other four tools all follow the three-way merge principles. Among them, Mastery reports the fewest 7,057 conflicting blocks and the fewest 3,782 conflicting scenarios. Among the 1,556

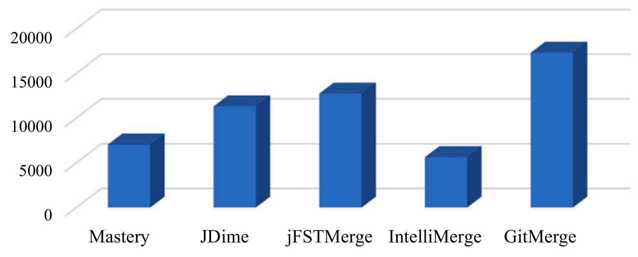


Fig. 10. The numbers of conflicting blocks of the five tools.

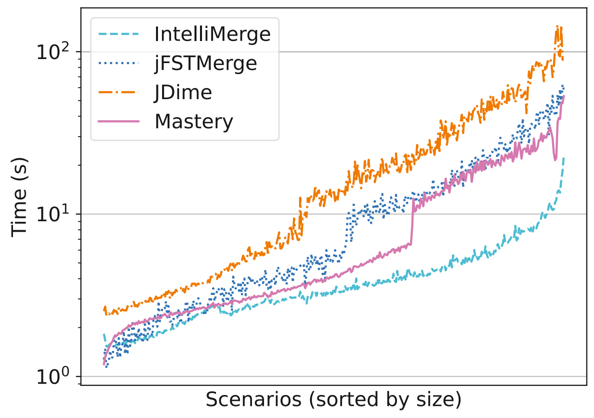


Fig. 11. Time cost of merging.

scenarios where JDime's results are conflicting whereas Mastery's are not, we find 52.76% involves shifted code—14.23% higher than the overall frequency.

7.6. Runtime performance (RQ4)

In this evaluation, we evaluate the performance from the perspective of runtime. As unstructured GitMerge is inherently particularly efficient, we only compare the runtime performance among semistructured and structured tools. Note that this comparison is fair as all these four tools are written in Java; they are executed on JVM under the same environment. Ignoring the failed runs, Fig. 11 shows the runtime on merge scenarios sorted by the size (i.e., total file size in unit of byte) of merge scenarios in ascending order. Since the runtime of each tool has considerable ups and downs even on the merge scenarios of a similar size, for clearer illustration, we plot each point as the average of adjacent 100 merge scenarios. The average times for the four tools are:

Mastery	JDime	jFSTMerge	IntelliMerge
10.16 s	25.11 s	13.73 s	4.30 s

Mastery is about 2.5 x as fast as JDime, and about 1.4 x as fast as jFSTMerge, which shows Mastery, as a structured merging tool, has competitive efficiency to semistructured merging tools.

7.7. Discussions

Threats to validity. We use expected versions, i.e. source files in merge commits, as ground truth. However, there are three threats to this ground truth:

1. There is no positive ground truth in merge commits since developers do not manually produce conflicts, but always try to resolve every conflict. But the resolving of conflicting changes in left and right versions must rely on additional information, which is usually available to a merging tool. Because of the lacking of positive ground truth, we cannot automatically recognize true positives, false positives and false negatives. Only true negatives can be automatically recognized, which are exactly the expected results in our taxonomy.
2. The expected version may not exactly be the merged version but the one postponed by a few commits, a.k.a. supplementary commits. Our dataset extraction process does not consider such supplementary commits. As a future direction, it is interesting to investigate how to obtain better ground truth by taking the supplementary commits into account (such as [21,22]).

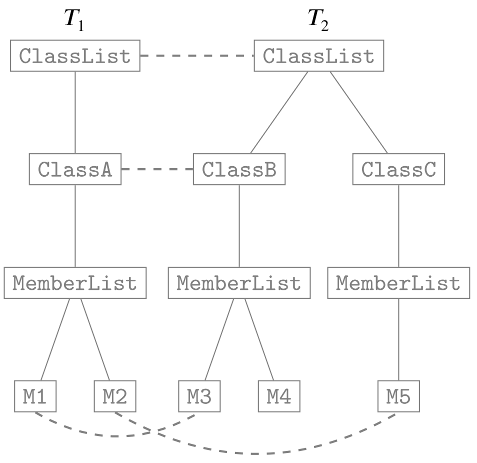


Fig. 12. A non-monotone matching between T_1 and T_2 (dashed lines show mappings).

3. As discussed in Section 7.3, because GitMerge is the default merging tool that developers use, if GitMerge reports no conflicts, its merging results will usually become the ground truth without a careful review by developers, even if the merging results are actually wrong.
4. By empirical inspection, we found developers introduce additional changes in some merge scenarios, rather than only collaborating on the changes from $left$ and $right$.

The ideal ground truth is to only merge changes in $left$ and $right$ correctly without introducing additional changes, and conflict blocks get reported if the changes are indeed semantically contradicted. Unfortunately, manual efforts seem inevitable approaching this ideality.

Limitations. Among the 1,109 merge scenarios where Mastery produces unexpected results while JDime produces expected results, we manually studied 10 random samples. We found that: In 4 merge scenarios, the expected versions introduce additional changes by developers or break three-way merge principles in other ways. Mastery produces the desired merge results w.r.t. three-way merge principles. The other 6 scenarios failed due to our limited support for *two-way merging*, where a merge scenario consists of only the two variants but not the base version. JDime realizes some heuristic two-way merging strategies, which can handle these merge scenarios better. These strategies can be realized in Mastery in the future.

Another limitation: Non-monotone matching. Generally but practically rarely, there could be non-monotone matching that obeys the original intuition of what matching is: to align nodes that should be the same one in the developing. For example, a member could be moved from one method to another method in development, which means the ancestor-descendant relationship between the statement and the methods is changed. However, even without such correct matching, it is also possible to correctly merge, which we will show by the example of Fig. 12.

Fig. 12 presents a non-monotone matching that can be produced by GumTree. The dashed lines plot the established mappings between T_1 and T_2. We see two class definitions ClassA and ClassB are matched. However, not all members of ClassA are mapped to members of ClassB – M2 is mapped to M5, which is a member of ClassC. By definition, we see the matching is not monotone. Such a matching brings troubles to AST amalgamation: in which class should the merge result of integrating M2 and M5 be on the target AST? It could be a member of ClassA (ClassB), or ClassC, and without further information, we cannot tell which is expected. Suppose we let T_1 be *base* and T_2 be a variant, one may interpret the mapping (M2, M5) as M2 is removed from ClassA and then inserted into ClassC. In that situation, M5 is a change and thus should be integrated as a member of `ClassC` on the target AST (if another variant does not make any inconsistent change) according to three-way merge principles. However, we claim the mapping (`M2`, `M5`) is not needed to achieve the same goal—without it, `M2` will still be regarded as a deletion (as it has no matches) and `M5` be regarded as an insertion (as it has no matches), according to rows 3 and 4 of Table 1. In this case, we have shown that there may be adorable to accept matches that are not monotone.

8. Related work

Westfechtel [23] and Buffenbarger [24] pioneered in proposing merge algorithms that exploit context-free and context-sensitive structures of programs. Language features such as alternatives, lists, and structures are represented at an abstract level. Later, differencing and merging approaches that are based on tree structures of programs have been widely explored.

Structured merge. Westfechtel [23] and Buffenbarger [24] pioneered in proposing merge algorithms that exploit structures of programs.

JDime [2] is a state-of-the-art tool for merging Java programs at AST level. In their AST representation, ordered and unordered lists are distinguished, and they propose distinct algorithms for merging them. We further distinguish ordered list nodes from constructor nodes (Section 2), as a list node can have an arbitrary number of children while a constructor node cannot. Their algorithm is in a top-down and level-wise manner and is unable to merge shifted code.

Later, two extensions of JDime are proposed. One is an auto-tuning technique that switches between structured and unstructured merge algorithms for better efficiency [3]; the other is a syntax-aware looking ahead mechanism for identifying shifted code and renaming in the AST matcher [12]. To be scalable, the lookahead mechanism has restrictions on the types of nodes when lookahead is enabled (an if- or try-statement), and the maximum search distance of lookahead (3 or 4). Note that in their work, the lookahead mechanism is not applied to merging. Unlike them, our merge algorithm efficiently handles shifted code in a general sense (i.e., without the above restrictions).

Asenov et al. [4] propose an algorithm for matching and merging trees using their textual encoding, which enables the usage of standard line-based version control systems. To yield precise matching, external information, for example, unique identifiers across revisions, is required. Unfortunately, they are directly unavailable. Furthermore, they have to perform expensive tree-matching algorithms.

Semistructured merge. Apel et al. [5] invented *semistructured merge* – a novel way of combining unstructured and structured approaches – that aims to balance the generality of unstructured merge and the precision of structured merge. Since semistructured approaches represent only part of the programs (typically high-level structures) as ASTs and keep the rest (low-level structures, e.g., method bodies) as plain text, they are not as precise as fully-structured approaches. An empirical study [25] on over 40,000 merge scenarios reveals that semistructured merge reports more false positives than structured merge.

Shen et al. [20] propose a graph-based refactoring-aware semistructured merging algorithm for Java programs, which is implemented as a tool IntelliMerge. The major difference between refactoring and shifted code is that refactoring must preserve semantics while shifted code usually does not.

Conflict resolution. Mens [1] thinks the resolution of conflicts caused by inconsistent changes made by variants is a major problem in version control. Since the resolutions of those conflicts are ambiguous, developers have the responsibility to resolve them manually. To alleviate manual efforts, Zhu and He [7] propose a synthesis-based technique that can automatically suggest candidate resolutions. In a real-time collaborative environment, it is also possible to simply prevent any presence of conflicts using locks [26–29].

9. Conclusion & future directions

We present Mastery, a three-way structured merge framework based on the methodology of combining the top-down and bottom-up visits of ASTs. This framework benefits from both the efficiency of handling trivial merge scenarios via a top-down pass and the effectiveness of handling non-trivial merge scenarios via a bottom-up pass. Experimental evaluations on real-world merge scenarios show that our approach achieves a higher merge precision and runs faster than JDime, a state-of-the-art structured merge tool. Moreover, Mastery can effectively merge shifted code, which reduces the presence of false conflicts.

In the future, we plan to support other programming languages in our framework, and further, improve the tree-matching and merging algorithms based on our evaluation findings.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002) and the National Key Research and Development Program of China (No. 2018YFB1308601).

References

[1] T. Mens, A state-of-the-art survey on software merging, IEEE Trans. Softw. Eng. 28 (5) (2002) 449–462, http://dx.doi.org/10.1109/TSE.2002.1000449.

[2] O. Leßenich, C. Lengauer, Adjustable Syntactic Merge of Java Programs (Master's thesis), Department of Informatics and Mathematics, University of Passau, 2012.

[3] O. Leßenich, S. Apel, C. Lengauer, Balancing precision and performance in structured merge, Automated Software Engineering 22 (3) (2015) 367–397, http://dx.doi.org/10.1007/s10515-014-0151-5.

[4] D. Asenov, B. Guenat, P. Müller, M. Otth, Precise version control of trees with line-based version control systems, in: M. Huisman, J. Rubin (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 152–169.

[5] S. Apel, J. Liebig, B. Brandl, C. Lengauer, C. Kästner, Semistructured merge: Rethinking merge in revision control systems, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 190–200, http://dx.doi.org/10.1145/2025113.2025141, URL http://doi.acm.org/10.1145/2025113.2025141.

[6] G. Cavalcanti, P. Borba, P. Accioly, Evaluating and improving semistructured merge, Proc. ACM Program. Lang. 1 (OOPSLA) (2017) 59:1–59:27, http://dx.doi.org/10.1145/3133883.

[7] F. Zhu, F. He, Conflict resolution for structured merge via version space algebra, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) http://dx.doi.org/10.1145/3276536.

[8] F. Zhu, F. He, Q. Yu, Enhancing precision of structured merge by proper tree matching, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings, ICSE-Companion, IEEE, 2019, pp. 286–287.

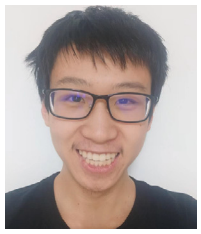
[9] J.R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 313–324, http://dx.doi.org/10.1145/2642937.2642982, URL http://doi.acm.org/10.1145/2642937.2642982.

[10] B. Fluri, M. Wuersch, M. PInzger, H. Gall, Change distilling:Tree differencing for fine-grained source code change extraction, IEEE Trans. Softw. Eng. 33 (11) (2007) 725–743, http://dx.doi.org/10.1109/TSE.2007.70731.

[11] G. Dotzler, M. Philippsen, Move-optimized source code tree differencing, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, New York, NY, USA, 2016, pp. 660–671, http://dx.doi.org/10.1145/2970276.2970315, URL http://doi.acm.org/10.1145/2970276.2970315.

[12] O. Leßenich, S. Apel, C. Kästner, G. Seibt, J. Siegmund, Renaming and shifted code in structured merging: Looking ahead for precision and performance, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, 2017, pp. 543–553, http://dx.doi.org/10.1109/ASE.2017.8115665.
[13] K. Zhang, T. Jiang, Some MAX SNP-hard results concerning unordered labeled trees, Inf. Process. Lett. 49 (5) (1994) 249–254, http://dx.doi.org/10.1016/0020-0190(94)90062-0, URL http://dx.doi.org/10.1016/0020-0190(94)90062-0.
[14] R. Tarjan, Depth-first search and linear graph algorithms, in: 12th Annual Symposium on Switching and Automata Theory, Swat 1971, 1971, pp. 114–121, http://dx.doi.org/10.1109/SWAT.1971.10.
[15] F. Zhu, X. Xie, D. Feng, N. Meng, F. He, Mastery: Shifted-code-aware structured merging, in: Dependable Software Engineering. Theories, Tools, and Applications: 8th International Symposium, SETTA 2022, Beijing, China, October 27-29, 2022, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2022, pp. 70–87, http://dx.doi.org/10.1007/978-3-031-21213-0_5.
[16] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, Change detection in hierarchically structured information, SIGMOD Rec. 25 (2) (1996) 493–504, http://dx.doi.org/10.1145/235968.233366.
[17] M. Pawlik, N. Augsten, RTED: A robust algorithm for the tree edit distance, Proc. VLDB Endow. 5 (4) (2011) 334–345, http://dx.doi.org/10.14778/2095686.2095692.
[18] S. Khanna, K. Kunal, B.C. Pierce, A formal investigation of Diff3, in: Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS '07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 485–496.
[19] A.B. Kahn, Topological sorting of large networks, Commun. ACM 5 (11) (1962) 558–562, http://dx.doi.org/10.1145/368996.369025.
[20] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, Q. Wang, IntelliMerge: A refactoring-aware software merging technique, Proc. ACM Program. Lang. 3 (OOPSLA) (2019) http://dx.doi.org/10.1145/3360596.
[21] T. Ji, J. Pan, L. Chen, X. Mao, Identifying supplementary bug-fix commits, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC, 01, 2018, pp. 184–193, http://dx.doi.org/10.1109/COMPSAC.2018.00031.
[22] T. Ji, L. Chen, X. Yi, X. Mao, Understanding merge conflicts and resolutions in git rebases, in: 2020 IEEE 31st International Symposium on Software Reliability Engineering, ISSRE, 2020, pp. 70–80, http://dx.doi.org/10.1109/ISSRE5003.2020.00016.
[23] B. Westfechtel, Structure-oriented merging of revisions of software documents, in: Proceedings of the 3rd International Workshop on Software Configuration Management, SCM '91, ACM, New York, NY, USA, 1991, pp. 68–79, http://dx.doi.org/10.1145/111062.111071, URL http://doi.acm.org/10.1145/111062.111071.
[24] J. Buffenbarger, Syntactic software merging, in: J. Estublier (Ed.), Software Configuration Management, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 153–172.
[25] G. Cavalcanti, P. Borba, G. Seibt, S. Apel, The impact of structure on software merging: Semistructured versus structured merge, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2019, pp. 1002–1013, http://dx.doi.org/10.1109/ASE.2019.00097.
[26] C.-W. Ho, S. Raha, E. Gehringer, L. Williams, Sangam: A distributed pair programming plug-in for eclipse, in: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology EXchange, eclipse '04, Association for Computing Machinery, New York, NY, USA, 2004, pp. 73–77, http://dx.doi.org/10.1145/1066129.1066144.
[27] S. Salinger, C. Oezbek, K. Beecher, J. Schenk, Saros: An eclipse plug-in for distributed party programming, in: Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 48–55, http://dx.doi.org/10.1145/1833310.1833319.
[28] M. Reeves, J. Zhu, Moomba – a collaborative environment for supporting distributed extreme programming in global software development, in: J. Eckstein, H. Baumeister (Eds.), Extreme Programming and Agile Processes in Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 38–50.
[29] H. Fan, C. Sun, Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 737–742, http://dx.doi.org/10.1145/2245276.2245417.

Fengmin Zhu is currently studying for his Ph.D. degree in Saarland University and CISPA Helmholtz Center for Information Security, Germany. He received a B.E. degree in computer science and technology in 2017, and an MSE degree in software engineering in 2020, both from Tsinghua University, China. He joined Max Planck Institute for Software Systems for a while, working on C program verification using Coq. His research interests include merging, testing, typing, synthesis, and verification.



Xingyu Xie is currently pursuing a master's degree in software engineering in Tsinghua University. He received a B.E. degree in computer science and technology in 2021, also from Tsinghua University. He is interested in logic and formal methods.

Dongyu Feng received his B.E. degree in software engineering in 2022, from Tsinghua University, China.

Na Meng is an associate professor in the Department of Computer Science at Virginia Tech, U.S. (since 2015). She received her PhD in Computer Science at The University of Texas at Austin, U.S. (2014). Her research interests include Software Engineering and Programming Languages. She focuses on conducting empirical studies on software bugs and fixes, and investigating new approaches to help developers comprehend programs and changes, to detect and fix bugs, and to modify code automatically. Nowadays, Dr. Meng also explores to fix security bugs automatically. Dr. Meng received the NSF CAREER Award in 2019.

Fei He is an associate professor at the School of Software of Tsinghua University. He received the PhD. degree from Tsinghua University in 2008. His research interests include model checking, program verification and automated logic reasoning. He has published over 80 papers in academic journals and international conferences. He is currently on the editor board of "Theory of Computing Systems" and "Frontiers of Computer Science". He has served as the PC member for many formal methods conferences, including ICSE, ESEC/FSE, CONCUR, FMCAD, SAT, ATVA, APLAS, ICECCS, SETTA, etc.